IN-61-CR

7601

p19

THE XPRESS TRANSFER PROTOCOL (XTP) A TUTORIAL (Short Version)

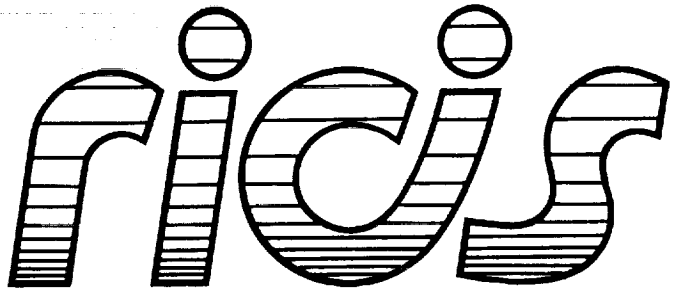
Robert M. Sanders
Alfred C. Weaver

Digital Technology

January 1990

Cooperative Agreement NCC 9-16
Research Activity No. SE.31

NASA Johnson Space Center
Engineering Directorate
Flight Data Systems Division



Research Institute for Computing and Information Systems
University of Houston - Clear Lake

T·E·C·H·N·I·C·A·L R·E·P·O·R·T

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information systems in 1986 to encourage NASA Johnson Space Center and local industry to actively support research in the computing and information sciences. As part of this endeavor, UH-Clear Lake proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a three-year cooperative agreement with UH-Clear Lake beginning in May, 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The mission of RICIS is to conduct, coordinate and disseminate research on computing and information systems among researchers, sponsors and users from UH-Clear Lake, NASA/JSC, and other research organizations. Within UH-Clear Lake, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business, Education, Human Sciences and Humanities, and Natural and Applied Sciences.

Other research organizations are involved via the "gateway" concept. UH-Clear Lake establishes relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research.

A major role of RICIS is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. Working jointly with NASA/JSC, RICIS advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research, and integrates technical results into the cooperative goals of UH-Clear Lake and NASA/JSC.

THE XPRESS TRANSFER PROTOCOL (XTP) A TUTORIAL (Short Version)

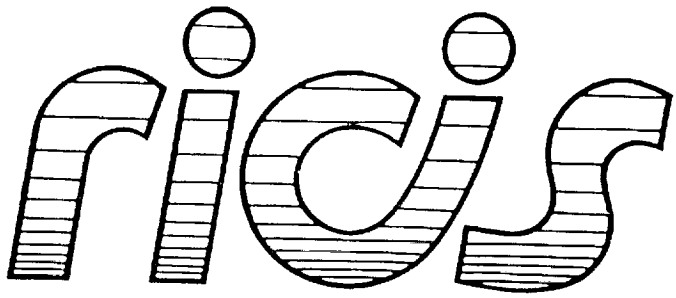
Robert M. Sanders
Alfred C. Weaver

Digital Technology

January 1990

Cooperative Agreement NCC 9-16
Research Activity No. SE.31

NASA Johnson Space Center
Engineering Directorate
Flight Data Systems Division



Research Institute for Computing and Information Systems
University of Houston - Clear Lake

T·E·C·H·N·I·C·A·L R·E·P·O·R·T

Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Robert M. Sanders, Alfred C. Weaver and Digital Technology. Dr. George Collins, Associate Professor of Computer Systems Design, served as RICIS technical representative for this activity.

Funding has been provided by the Engineering Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Frank W. Miller, of the Systems Development Branch, Flight Data Systems Division, Engineering Directorate, NASA/JSC.

The Xpress Transfer Protocol (XTP) — A Tutorial

Robert M. Sanders and Alfred C. Weaver

Computer Networks Laboratory
Department of Computer Science
University of Virginia

Introduction

XTP is a reliable, real-time, light weight *transfer*[1] layer protocol being developed by a group of researchers and developers coordinated by Protocol Engines Incorporated (PEI).[1,2,3] Current transport layer protocols such as DoD's Transmission Control Protocol (TCP)[4] and ISO's Transport Protocol (TP)[5] were not designed for the next generation of high speed, interconnected reliable networks such as FDDI and the gigabit/second wide area networks. Unlike all previous transport layer protocols, XTP is being designed to be implemented in hardware as a VLSI chip set. By streamlining the protocol, combining the transport and network layers and utilizing the increased speed and parallelization possible with a VLSI implementation, XTP will be able to provide the end-to-end data transmission rates demanded in high speed networks without compromising reliability and functionality. This paper describes the operation of the XTP protocol and in particular, its *error*, *flow* and *rate* control, inter-networking addressing mechanisms and multicast support features, as defined in the XTP Protocol Definition Revision 3.4.[1]

Future computer networks will be characterized by high reliability and very high data transmission rates. Traditional transport layer protocols, such as TCP and TP4, which were designed in an era of relatively slow and unreliable interconnected networks, may be poorly matched for the emerging environment. Although they contain many necessary features, such as error detection, retransmission, flow control and data resequencing, they are deficient in many respects — they do not provide rate control and selective retransmission, reliable multicast is not supported, their packet formats are complex and require extensive parsing due to variable header lengths and support of complex modes. These protocols manage many timing events at both the sender and the receiver — for example, since the sender does not initiate receiver data acknowledgements, both the receiver and sender require an additional timer. The data transmission rates assumed are no longer valid and may limit the scalability of the protocols — in TCP, for example, which was designed in an era of 56Kbps data transmission rates, the flow window size is small, and based on 16 bit byte sequencing. Finally, the state machines for these transport protocols were intended for sequential rather than parallel execution. For example, the placement of the checksum field was considered arbitrary and so it was placed in the header.

XTP provides for the reliable transmission of data in an inter-networked environment, with real-time processing of the XTP protocol — i.e., the *processing* time for incoming or outgoing packets is no greater than *transmission* time. XTP contains error, flow and rate control mechanisms similar to those found in other more modern transport layer protocols[2] in addition to multicast capability. Timer management is minimized — in XTP there is only one timer at the receiver, used in closing the context. XTP has a 32 bit flow window. XTP's state

1. The *transfer* layer is formed by combining the functionalities of both the network and transport layers of the ISO OSI model into a single layer.

2. Specifically, two other modern transport layer protocols — Versatile Message Transaction Protocol (VMTP) developed at Stanford University by David Cheriton, and Network Bulk Transfer (NETBLT) developed at MIT by David Clark.

machine is specifically designed for parallel execution. Address translation, context creation, flow control, error control, rate control and host system interfacing can all execute in parallel.

The XTP protocol is considered a *lightweight* protocol for several reasons. First, it is a fairly simple yet flexible algorithm. Second, packet headers are of fixed size and contain sufficient information to screen and steer the packet through the network. The core of the protocol is essentially contained in four fixed-sized fields in the header — KEY, ROUTE, SEQ and the command word. Additional mode bits and flags are kept to a minimum to simplify packet processing.

Types of XTP PDUs

XTP utilizes two frame formats, one for *control* packets and one for *information* packets (see Figure 1).

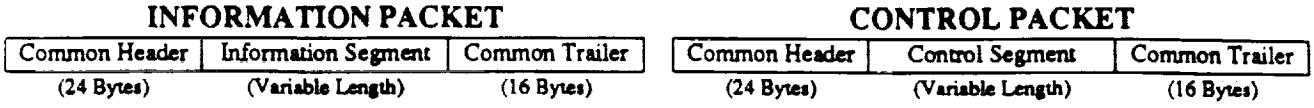


Figure 1. General Frame Formats

Both formats share a common *header* segment and a common *trailer* segment, each of constant length. Each XTP packet includes a variable length segment between the header and trailer whose segment type determines the packet type. The important fields are aligned on 8 byte boundaries so that they can be quickly accessed by any machine with 2 byte, 4 byte or 8 byte alignment. The formats are described in greater detail later.

The common *header* specifies the packet type and identifies what portion of the data stream, if any, is included in the information segment. Optional modes, such as disabling error checking or multicast transmission, are indicated in the packet header's *control flags* field. The common *trailer* contains two checksum fields, identifies how much of the data stream has been delivered to the receiving client application, and also contains a *flags* field. These flags generally control state changes, for example closing the data transmission connection or requesting data acknowledgement. Message boundaries are also specified in the trailer by setting the *end of message* flag (EOM).

The *information* segment contains the user data being transferred, and is also used to pass addresses and other miscellaneous data when appropriate. Each *data* packet contains a contiguous subset of the data stream being transferred. In XTP, there is no protocol-imposed upper limit on the number of bytes included in each data packet — each implementation is bounded by the underlying datalink layer. For each implementation this limit is known as the *maximum transmission unit* (MTU) and is found by subtracting the XTP header and trailer sizes from the datalink's maximum data field size. XTP supports two additional modes of data transfer which allow out-of-band, tagged data of constant length (8 bytes) to be included in the data packet along with the user's data. These additional data bytes also appear in the *information* segment, either at the beginning or at the end of the usual user data. Their presence is indicated by flags in the header and trailer (the *tag*). Beginning tagged data are indicated by the BTAG flag in the common header. Ending tagged data are specified with the ETAG flag in the common trailer.

The *control* segment contains the receiver's *error*, *flow* and *rate* control parameters' values. This segment also contains fields used to resynchronize the transmitter and receiver when necessary.

Multi-Packet Handshaking

In XTP, multi-packet exchange sequences provide user applications with both a transport-level *virtual circuit* capability and a transport-level *datagram* service. For example, in XTP a connection may consist of an exchange of three packets, as shown in Figure 2.

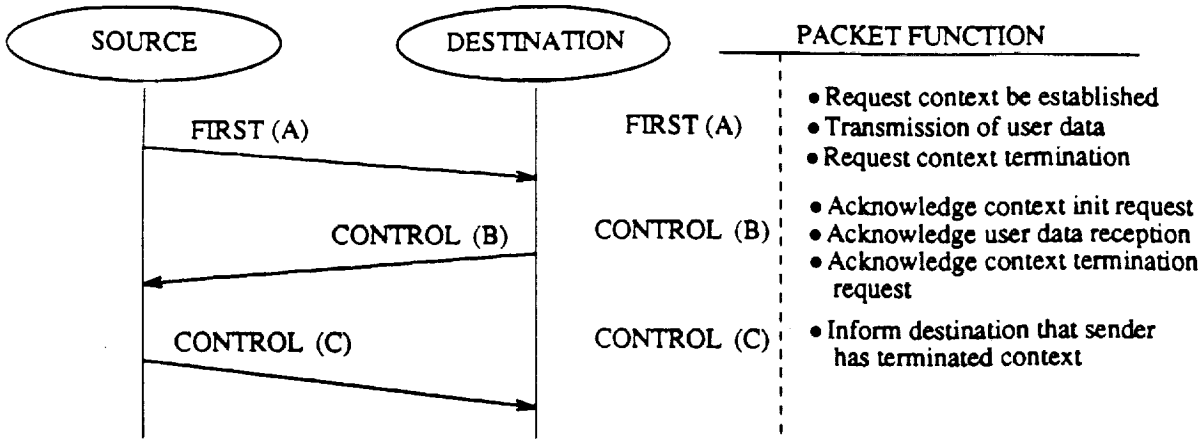


Figure 2. Three Packet Connection-Mode Handshake

The scenario above depicts how XTP can reliably set up a connection between two user processes, transmit data, and close the connection with a minimum of three packets. In this scenario, the source initially transmits packet (A). At the destination the header is examined and it is determined that the source wishes to establish a send connection. If the destination wishes to comply, a context is established. The packet's data are then queued for transfer to the waiting destination user process.

After successfully transferring the received data to the host, the destination complies by sending control packet (B). This packet acknowledges the receipt of the data, and indicates that the destination is also ready to close the connection. On receiving packet (B), the sender emits control packet (C), and closes its side of the connection — thus completing the three way handshake. Any buffers still associated with the connection are freed, and the sender will no longer respond to control packets arriving for the context. When packet (C) is received by the destination, the connection is closed.

In XTP the burden of detecting lost acknowledgements is assummed by the sender. The sender requests acknowledgement by setting the *status request* bit (SREQ) in the XTP common trailer. A timer (WTIMER) is used by the sender to determine if the receiver has failed to respond to a sender-generated request for current status and data acknowledgement. If the timer expires before an acknowledgement arrives, the sender assumes the acknowledgment was lost, and sends another request for a control packet acknowledging the received data. On the other hand, when closing, the source acknowledges context termination, so that the receiver can be sure that the context is closed. If this last packet gets corrupted or lost, the receiver will eventually timeout and close the connection.

Unlike TCP, where each data packet would be retransmitted after the timeout, in XTP only a CNTL packet containing the SREQ would be sent. The corresponding returned CNTL packet would indicate which data packets, if any, to retransmit. This is a conservative procedure which forces a "synchronizing handshake" before retransmitting except when retransmission is explicitly indicated by the receiver.

Closing an XTP connection is coordinated using the three flags RCLOSE, WCLOSE and END. The local host sets the RCLOSE or WCLOSE flags in an out-going packet to inform the remote host that it has completed all reading or writing it intends to perform on the shared connection. Note that in a full duplex connection between two nodes A and B data would be transmitted in both directions ($A \rightarrow B$ and $B \rightarrow A$). Using RCLOSE and WCLOSE, each direction can be shut down independently.

The END flag is set in an outgoing packet to signal to the remote host that the local host released or closed its end of the connection. Thus, END is set in the final packet transmitted, and indicates that the context has been terminated — i.e., that it is gauranteed that no further packets can be exchanged. If, at any time, a packet is received with the END bit set, the context is assumed closed at the remote end, and the local host releases the

context.

The "close" protocol based on END, RCLOSE and WCLOSE can uniformly support the three packet graceful termination of Figure 2, an abbreviated termination, transactions, and abort situations without modification.

Two-packet, transaction-like packet exchange sequences are also allowed in XTP and are referred to as *fast handshakes*. For *full duplex* connections, these modes are less reliable than the three packet connection. The two packet fast close can be considered a transport level datagram service or the basis for simple request/response operations.

Note that when a *data* acknowledgment is requested in XTP, as in the FIRST packet of Figure 2 (packet A), the acknowledgment is not necessarily provided immediately. In the fast close cases, the receiver delays acknowledgment until all data received prior to the SREQ have been processed. This includes the data contained in the packet with SREQ.

XTP contains a second status request flag in the common header flags field which is called DREQ. DREQ differs from SREQ in that SREQ requests a response immediately from the receiver, and DREQ requests it after the currently queued data have been received at the receiver. This is useful because the acknowledgement is delayed until the receiver has freed the buffer space associated with the queued data and is capable of accepting more data from the sender. Flow control blocking is minimized.

In closing, SREQ behaves like DREQ — if this were not so, the receiver might generate its final acknowledgement packet before the data from the last information packet has been delivered to the receiving client. But the sender could not close the context since unacknowledged data exists. Thus, in closing, SREQ responses are delayed until all data have been processed.

Flow Control

Flow control allows the receiver to inform the sender about the current state of its receiving buffers. In XTP, the receiver's flow control parameters are included in control packets sent from the receiver to the sender. These parameters are shown in Table 1.

Parameter	Type	Location	Description
ALLOC	32 bit sequence number	control segment	1 + sequence number of last byte receiver will accept.
DSEQ	32 bit sequence number	common trailer	1 + sequence number of last byte receiver delivered to destination client process.
RSEQ	32 bit sequence number	control segment	1 + sequence number of last byte receiver accepted.
ALLOC - DSEQ	Size of receiver's data buffer in *bytes*.		
RSEQ - DSEQ	Number of bytes received and waiting to be transferred to destination client process.		

TABLE 1. XTP Flow Control Parameters

ALLOC constrains the sender from introducing more data than the receiver's buffers can accept. The sender refrains from sending bytes with sequence number ALLOC or higher. Thus, ALLOC is one greater than the highest byte sequence number that the receiver will accept. DSEQ is the sequence number of the next byte to be delivered to the destination application process, or client. Likewise, DSEQ can be thought of as one greater than the sequence number of the last byte delivered to the destination client. All bytes with sequence number less than DSEQ have been successfully transferred to the destination client. DSEQ is always less than or equal to ALLOC. Subtracting DSEQ from ALLOC (modulo 2^{32}) yields the buffer size allocated in bytes to the context by the receiving XTP process. Thus, XTP uses a *byte-oriented* allocation scheme as in TCP instead of a *packet-oriented* allocation scheme as in TP4. An advantage to this policy is that byte-oriented allocation is not affected by internet fragmentation.

The sender holds data that have been transmitted in a buffer until it knows the data have been delivered to the destination client. As long as the data are buffered, it can be retransmitted if necessary. When the sender notes

that DSEQ has been extended, it frees the buffers associated with the delivered data.

RSEQ is the sequence number of the first byte not yet received contiguously from the network. This can be the first byte in the first gap, or the first byte in the next data packet expected. As with ALLOC and DSEQ, an alternative interpretation exists for RSEQ. All bytes associated with sequence numbers less than RSEQ have been buffered by the receiving XTP process at the destination, but may not have been delivered to the destination client process yet. Thus, RSEQ is one greater than the largest *consecutively* received data byte sequence number. The sequence numbers of all bytes associated with gaps lie between RSEQ and ALLOC.

Collectively, these parameters provide the means for XTP to implement *flow* control whereby the receiver can restrict the sender from sending excessive data prematurely. Note that all sequence number parameters in XTP occupy 4 bytes — SEQ, RSEQ, DSEQ, ALLOC and the sequence number pairs contained in the SPAN field of CNTL packets associated with gaps in the received data stream.

It should be observed that each byte still in the *bit pipe*, i.e., each byte currently in transit or still subject to retransmission, must be uniquely identifiable, so that retransmission is possible. In TCP/IP sequence numbers are limited to 16 bit numbers with only $2^{16} = 64K$ bytes possible in the bit pipe at any given point in time. On the other hand, XTP's 2^{32} bit patterns yield over 4 billion unique sequence numbers. Thus, XTP is more naturally suited to networks with both high bandwidth and/or high end-to-end latency than TCP/IP.[3]

Once the sender has been informed of the receiver's allocation limit via the ALLOC parameter, it continues to transmit until the allocation has been reached, without the need for individual acknowledgements of each packet transmitted. Thus, XTP more efficiently utilizes the higher reliability of modern networks, such as fiber optic LANs. Once the allocation has been reached, the XTP sender process sets the SREQ parameter in the last data packet transmitted, and the receiver responds as earlier described with a control packet that acknowledges all data received, describes any gaps detected, and, if appropriate, advances the allocation.[4]

An alternative allocation policy exists in XTP based on the size and availability of the receiving client application's buffers. This mode is referred to as *reservation* mode. In reservation mode, the transmission is determined by the size of the receiving user's buffers reserved specifically for the context (by setting ALLOC to the size of this reserved buffer). In this mode, the sender must pause between message transmissions until the receiving client has posted a new client buffer to receive the next message. This is necessary to separate adjacent messages into different client buffers, since each message may not entirely fill its buffer.

In reservation mode, the reservation buffer size may differ greatly from the normal allocation size, and may be greater. This mode is similar to the the allocation control mechanisms in the VMTP[7] and NETBLT[8] protocols.

Rate Control

In some situations flow control is not sufficient to ensure efficient, error-free transmission between the sender and receiver, even on an extremely reliable network. Imagine a network containing both hardware and software implementations of the XTP protocol. Since the VLSI chip set will allow much of the protocol to be executed in parallel, a sending XTP process implemented in hardware may overwhelm a receiving XTP process implemented in software if it sends multiple, back-to-back packets.

---

3. Van Jacobsen has proposed extending the TCP protocol to, among other things, include 29 bit sequence numbers to extend the size of the TCP flow control window.[6]

4. Note that other policies are possible for determining when to set the SREQ bit in XTP; in XTP, the SREQ policy is determined by the user application.

XTP uses *rate* control to restrict the size and time spacing of bursts of data from the sender. Within any small time period, the number of bytes that the sender transmits must not exceed the ability of the receiver (or intermediate routers) to decipher and queue the data — otherwise they will be overwhelmed and begin dropping packets, creating gaps in the received data stream. This problem is independent of the flow control/buffer size problem discussed previously. The receiver may have adequate buffer space available, but back-to-back packets may arrive faster than the XTP receiver process can analyze them. The XTP parameters used to implement rate control are shown in Table 2. Together, the two rate control parameters allow the receiver to tune the data transmission rate to an acceptable level.

Parameter	Location	Description
RATE	control segment	Maximum number of bytes receiver will accept in each one second time period.
BURST	control segment	Maximum number of bytes receiver will accept per burst of packets. The transmitter may not transmit more than BURST bytes between RTIMER timeouts.
RATE/BURST	Maximum number of packet bursts per second.	
BURST/RATE	Seconds per Packet Burst. The rate timer (RTIMER) is set to this value.	
RATE = -1	Rate control is disabled — i.e., sender transmissions are unconstrained.	

TABLE 2. XTP Rate Control Parameters

When a slow XTP receiver implemented in software is listening to a hardware-implemented sender, packet bursts must be time spaced to guarantee that the slow receiver has sufficient time between back-to-back packet bursts to complete protocol processing before the arrival of the next burst. With the above parameters, inter-packet spacing can be achieved as follows. Set the BURST parameter equal to the MTU (maximum transmission unit) of the underlying network. Thus, each packet "burst" may not contain more than one packet's worth of data. If the receiver can handle *N* packets per second, set RATE equal to *MTU* * *N*. In this manner, the sender is constrained to spacing back-to-back packets accordingly. See Figure 3 which plots bytes transmitted versus time during a one second time period for a hypothetical XTP transmitter.

The RATE and BURST parameters are adjustable, and for each implementation of XTP, appropriate values could be determined experimentally. Their values would then be included in all out-going control packets from the receiver. Note that in this example, RATE > > BURST.

In Figure 3, the BURST and RATE parameters have been adjusted such that an inter-burst *separation* occurs. Each burst of data is depicted by a ramped triangle. The separations between adjacent bursts are shown by horizontal dotted line segments in which no progress is made towards the top of the graph. During each pause in the transmitter, the slower receiver is allowed to catch up.

Unfortunately, the sender process does not know the appropriate RATE and BURST values to use with a particular receiver until the first burst of data has been completed; the proper value for ALLOC is also unknown initially. The appropriate values only become known when the first control packet arrives at the sender. Before this control packet is returned, the sender must use default values for the various *flow* and *rate* control parameters.

If protocol processing speeds vary widely on a network, the default values for ALLOC and *rate* control parameters affect the number of dropped packets during the initial data burst. A conservative approach would be for the sender to set the default ALLOC to a small number of bytes (say one average sized data packet as defined by the *maximum transmission unit)* and to use the aforementioned approach to setting the default *rate* parameters such that packet spacing is sufficient for the slowest receiver on the network. After the initial burst, which also establishes the context connection, the sender would block, waiting for the returned control packet generated by the SREQ in the last data packet of the burst. This control packet would contain the more accurate *flow* and *rate* control parameters specifically applicable to the receiver. In this case, few packets would be lost at the cost of moderately more overhead in the initial burst.

Consider a hardware-implemented router between an FDDI LAN and an Ethernet LAN that can absorb back-to-back packets as fast as they arrive, but has limited buffer space. Rate control can be used to avoid overrunning

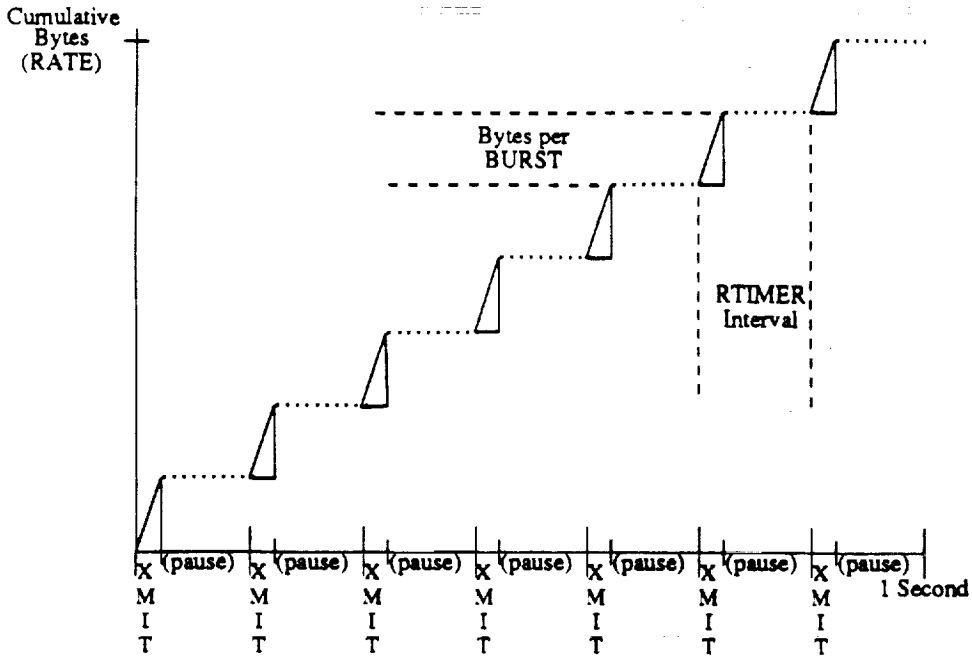


Figure 3. Rate Control of a Hypothetical XTP Transmitter

the router's buffers. To implement this, BURST could be set equal to RATE, and RATE would be set to the rate at which the router could relay frames for the context in terms of bytes per second. In this scenario, the RTIMER's interrupt rate would be once per second, and the number of bytes per second allowed would equal RATE. As more inter-network contexts become established, the router may need to restrict the burst rate for existing contexts with the RATE (=BURST) parameters. Later, as contexts become inactive or removed from the inter-network, the router may chose to increase the flow rate of the remaining contexts. RATE (=BURST) would be increased in outgoing control packets in this case. RATE and BURST allow the router to dynamically control the flow into the router so as to avoid overwhelming it with requests.

XTP's rate control feature may be disabled by setting RATE equal to -1 in outgoing CNTL packets.

Error Control

When errors do occur in transmission, XTP, like TCP and TP4, must detect the errors and initiate retransmission of the erroneous data. XTP uses two checksums over the XTP packet contents to verify the integrity of the data received through the network. These two checksums appear in Table 3. The XTP checksum algorithms were chosen for speed and VLSI compatibility; details of their operation are found in Appendix A of the XTP Protocol Definition version 3.4.[1]

It is preferable to place the checksums in the last few bytes of the XTP frame so that the checksum calculation can be concurrent with packet transmission or reception. If the checksums were placed in the front of the packet, the entire packet would have to be accessed to compute the checksum before packet transmission begins. Thus, two sweeps over the data would be necessary — one for the checksum, and one for copying the bytes to the network. This inefficient approach is inherent to TCP and TP4, whose checksums occur before the information segment, and avoided in XTP where the checksums follow the rest of the packet and are found in the common trailer.

When either checksum indicates that the packet received contains erroneous information, the receiver assumes the packet is garbled and discards it. If the source were known, the receiver could immediately inform the source XTP sender process that the packet was garbled in transit — allowing the source to begin retransmission.

Parameter	Location	Description
DCHECK Value	trailer (4 bytes)	4 byte checksum over data fields. Includes the control segment in control packets; the information segment in information packets.
HTCHECK Value	trailer (4 bytes)	4 byte checksum over header and trailer.
NODCHECK Flag	trailer *flags* field (1 BIT)	Flag used to signify that DCHECK checksum is not present in current packet.
NOCHECK Flag	header *flags* field (1 BIT)	Flag used to signify that checksum calculation is disabled in current packet.
XOR	Calculated using exclusive-OR operations only. Represents the vertical parity of data bytes.	
RXOR	Each intermediate result is left rotated before exclusive-ORing in the next word.	
XTP's checksum function is formed using left rotation and exclusive-OR operations over the 16-bit words covered. The 4 byte checksum is the concatenation of two 2-byte checksums XOR and RXOR. (XOR | RXOR).		

TABLE 3. XTP Checksum Parameters

Normally, this information is available by referencing the packet's KEY field, located in the common header, that uniquely identifies the originating client process at the node that transmitted the packet. But, the receiver cannot assume that the KEY field is correct, since the error could conceivably have occurred anywhere within the packet including the KEY field itself (if the HTCHECK checksum is invalid). Thus, the receiver always discards packets received with errors.

At the sender, transmission continues as if no error had occurred. The next packet is placed onto the network. If this new packet arrives correctly, the receiver examines the starting sequence number for the packet. Like the context identifier KEY, the starting sequence number is contained in the packet's header (in the SEQ field). The receiver expects the SEQ value of the incoming packet to equal the current RSEQ value for the context. Since a packet was dropped, the incoming SEQ is larger than RSEQ by the size of the dropped packet. The receiver accepts the data packet, noting that it arrived out of sequence, and that a *gap* exists in the data stream. Now the receiver can utilize the KEY information of the current packet to send back a CNTL packet to announce the gap. Having the receiver indicate when a gap has been detected is optional in XTP; if the receiver fails to send the CNTL packet, the sender will eventually include a SREQ and block, or timeout.

Gaps And Selective Retransmission

A receiver could describe a gap using a pair of sequence numbers that bound the gap. Instead, XTP describes the groups of bytes (called *spans)* which *were* received. This process is known as *selective acknowledgement.* Thus, in XTP, the location of gaps is inferred to be between the spanning byte groups selectively acknowledged. Each byte group is described with two sequence numbers that bound the bytes received. The first sequence number in the pair marks the byte where the group started (i.e., the first byte *in* the group). The second sequence number is one greater than the last byte in the group (i.e., the first byte *not contained in* the group.) Between each pair of received byte groups is a gap, or hole, in the received byte stream encompassing one or more bytes.

XTP allows a receiver to track and notify up to 16 separate gaps for any given context. This capability is not required, however — receivers may chose to ignore all out-of-sequence data. In this case the receiver would never allow gaps to be created, and would force the sender to retransmit both lost data and correctly received out-of-sequence data. This latter method is referred to as *go-back-n* retransmission. Since up to 16 byte groups may be described in any CNTL packet, the SPAN field, which contains descriptors for the gaps, is variable in length.

Each gap spans a portion of the data stream. For 16 individual gaps to accumulate would presumably be a rare occurrance, and only possible when large volumes of data are transmitted with few SREQs. Consider a massive file transfer between mainframes with considerable buffer space. The entire file could be transferred with a single SREQ in the final data packet. Any lost data could be determined and communicated to the sender in a minimum number of CNTL packets (one) in most cases. This process, in which only the lost data are retransmitted, is

known as *selective retransmission.*

XTP Timing Considerations

Typically, *connection* timers are used to detect the possibility that a connection has been severed. In XTP, the CTIMER is used to monitor for such events. CTIMER expires when the connection has been inactive for 60 seconds. By the time a break is suspected, a number of attempts may have been made to prompt the other "end" to re-synchronize the protocol. In XTP, these prompts are in the form of CNTL packets (called *sync* packets). If, after a number of attempts have been made, the situation has not improved, the XTP process will inform its client application process of the situation, and if so directed, close the connection.

Re-synchronization is attempted when the sender has issued a SREQ to the receiver and the WTIMER times out before the receiver's CNTL packet has been received by the sender, as earlier described. The XTP sender process will assume that the packet containing the SREQ was dropped, and transmits another packet containing an SREQ to the receiver — a *sync* packet.

XTP associates each receiver-generated CNTL packet with the SREQ that requested it. When the sender issues a *sync* packet, it increments a counter value (the SYNC counter for the context), and includes this value in the SYNC field of the outgoing *sync* packet. When the receiver receives *sync* packets from the sender, it copies the SYNC value from the incoming CNTL packet into the ECHO field of the outgoing CNTL (called an *echo* packet). The sender differentiates between old *echo* packets and the current one by comparing the ECHO value against the current SYNC counter contents.

When an incoming ECHO matches the context's SYNC counter value, the sender examines the receiver's current status data. If no retransmissions are needed, and ALLOC has been extended, the sender resumes with data transmission. If the receiver has not extended ALLOC, but there are gaps to retransmit, the sender begins retransmitting the lost data. Otherwise, the sender must wait for the receiver to extend the ALLOC value before proceeding, and must block.

Sync/echo packets are also used to update the current round trip time (RTT) estimate. The sender sets the TIME field in the *sync* packet to the current time at the sender. When the receiver prepares the corresponding *echo* packet, it also copies the TIME field of the *sync* packet into the TIME field of the *echo* packet. When the sender receives the *echo* packet, it estimates the current round trip time by subtracting the echoed TIME value from the current time. This RTT estimate is used by the sender in setting the duration of the WTIMER. WTIMER is set to twice the RTT. Since XTP acknowledgements are generated at the sender's request (using SREQ), the RTT estimate more accurately reflects the average round trip time than schemes relying on timeout-generated acknowledgements.

XTP bounds the time each packet is allowed to "live" in the network using the time-to-live (TTL) field. The time value is expressed in 10 millisecond "ticks". In outgoing packets, this field is initialized by the user to a given number of ticks (in TCP time-to-live is based on the current RTT estimate). At each hop, the TTL value for the packet is decremented — when the value becomes zero or negative, the packet has exceeded its time to live and is discarded. Note that bounding the time the packet can exist on the inter-network aids in removing packets which can not be delivered due to pathological situations such as host or router crashes.

Since the TTL field occupies 2 bytes, 64K different values are expressible in the field yielding a range in values from zero seconds to 655.36 seconds in 10 millisecond steps. For networks with greater propagation time than 655 seconds, (e.g., a very wide area network) the TTL mechanism must be disabled. XTP allows the TTL mechanism to be disabled by setting the initial TTL value to zero. If a packet arrives with a TTL value of zero, it is assumed that the policy is to bypass the TTL decrement-and-discard step, and the packet is relayed onto the next network with the TTL value still equal to zero.

When transmitting, the sender may use a timer to comply with the receiver's *rate* control requirements for bytes/second (RATE) and bytes/burst (BURST). This timer (RTIMER) must be accurate enough to support the rate control timing requirements for the given implementation. The duration of the RTIMER is set to BURST/RATE seconds. Each RTIMER timeout reestablishes the limit on the maximum number of bytes which can be output on the context during the next RTIMER time period.

XTP requires only one timer at the XTP receiver process, and it is only needed during connection closing. This timer is set whenever the receiver process issues a CNTL packet with the RCLOSE request set. The timer estimates the round trip time, and if necessary, generates a new RCLOSE request upon expiring.

Each multi-context route requires a special timer called a *Path* timer (PTIMER). In routers, the PTIMER duration may extend for days to allow datagram-type service over stable, infrequently used routes. In each end-node, the PTIMER duration is substantially shorter, and may be measured in minutes or hours.

Addressing Mechanisms In XTP

XTP was designed to interface with a variety of datalink layers. In each case, XTP packets must be encapsulated within the PDUs of the underlying datalink layer. For those protocols capable of multiplexing their services among multiple transport layers (say XTP and TCP simultaneously), the datalink layer uses a unique, standardized identifier to distinguish between TCP and XTP packets. In 1990, XTP is expected to be operational on Ethernet, IEEE 802.5, and FDDI; XTP is already operational on top of the User Datagram Protocol (UDP).

Within the XTP layer, each end of a XTP connection must be able to uniquely identify its peer. To complicate matters, XTP's inter-network and multicast capabilities impose additional addressing requirements.

Rather than including all relevant addressing data explicitly in each packet, XTP caches the addressing data contained in the first packet at both the sender and receiver, and uses the KEY field as a lookup index into the cache to access the actual addresses as needed. As described earlier, this initial packet is a special *information* packet of type FIRST. The following packets contain only the KEY, resulting in smaller packets because the KEY is encoded in fewer bytes. Thus the KEY field is included in every packet, and is located in the common header segment.

The KEY is generated by the node initiating the connection, and included in the FIRST packet transmitted to the receiver. Also included in this FIRST packet are addressing data used to identify the intended receiver. These addresses are contained in a list for comparison with the receiver's address filter. In multicast mode, more than one receiver is targeted for each packet. The appropriate receivers note the arrival of the FIRST packet, and save the context identifier (KEY), the source of the datalink frame containing the packet (MAC address) and the route identifier (ROUTE) in a database associated with the *context record*. Subsequent packets need not contain the destination network address since the triple <MAC,KEY,ROUTE> can be used to lookup the context.

The KEY field is 32 bits in length, but the context identifier KEY's value is restricted to a value expressible in 31 bits. The extra bit is located in the most significant bit position, and reserved for determining the direction of the packet — i.e., which end of the connection generated the packet. Packets *sent* from the node which generated the KEY value have the bit set to *zero;* packets *received* at the node generating the KEY value have the bit set to *one*. When the high bit is set, the KEY is referred to as a *return* key. If the KEY in an incoming packet's header is a return key, the receiver can use the key as a lookup to determine the context for an incoming packet since the receiver generated the original key.

In order to make context lookup faster, the receiver the receiver must be able to substitute a value of its own choosing for the newly forming context's KEY. But the new KEY will only be useful if the peer uses it when transmitting packets on this context. The substitute KEY is transmitted back to the context initiator in the XKEY field of the next CNTL packet. The receiving XTP context continues to *output* CNTL packets containing the original KEY (with the high bit set), whereas the sending XTP context will adopt the receiver's requested KEY

when transmitting packets (also with the high bit set). Note that in this case, once KEYs have been *exchanged*, all packets will be using *return* keys — with the high bit set. See Figure 10. Key exchanging is only possible when there is a unique receiver (i.e., keys may not be exchanged in *multicast* mode). After exchanging keys, any additional packets sent in either direction contain the appropriate *return* KEY value for the packet's destination.

The *address* segment included in the FIRST packet contains two main fields — a fixed length descriptor field indicating the addressing format used, and a variable length field containing the actual list of addresses. At present, compatible formats are supported for both Darpa Internet and ISO formats.[9] Formats for accommodating Xerox XNS[10] style addresses, U.S. Air Force Modular Simulator project (MODSIM) addresses and Source Route addresses are under study and should be available in future versions of the XTP protocol.

Address *filtering* occurs at the receiver when determining whether to establish the connection requested by the sender of a FIRST packet. Beforehand, the receiving client describes to the XTP receiver process the set of network addresses to which it will connect. When a FIRST packet arrives, the receiver compares its address segment contents against the receiving client's address filter to determine whether to accept the packet or not.

In the event that the network topology is known, and network addresses do not require more than 4 bytes, XTP can use a direct addressing mode. In this mode, the KEY field contains the actual destination address rather than an index used to look up the context. This direct addressing mode is invoked by setting the DADDR flag in each packet. The DADDR flag is located in the common header.

XTP Inter-Network Routing

When connecting to a process on a remote network, a connection must be established through one or more routers until the destination network is reached, and finally to the remote host on which the receiver client resides. The router receives the packet on the first network, makes a routing decision, and outputs the packet onto the second network. The router must be capable of determining the appropriate node on the new network to which the packet should be transferred, based on the destination address information contained in the packet. As in the single network case, this addressing information can be cached.

The ROUTE field serves a similar purpose to the KEY field and is utilized by the router to locate the proper addressing data in its cached address translation map for a packet traveling on a given route. As with the KEY, ROUTE values can be exchanged between adjacent routers and/or the endpoint nodes.

When a FIRST packet arrives at the router, the router saves the incoming ROUTE value in the data structure associated with the route upon which the packet is travelling. Packets generated at the router to be returned to the context initiator will use the *return* form of this ROUTE value. The Router has the option of generating its own ROUTE values for the next host or router in sequence to use on the given route. When relaying the FIRST packet towards the destination, the router merely substitutes its preferred ROUTE value in the header, overwriting the original ROUTE value chosen by the context initiator.

Packets arriving at the router from the destination-end of the connection will contain the *return* form of the router's desired ROUTE value. The destination-end of the connection may choose to exchange ROUTE values with the router. If so, it will set the XROUTE field to its chosen ROUTE value when transmitting its first CNTL packet. The router will note the XROUTE value, and use its *return* form in future packets to the destination-end.

The router relays the CNTL packet towards the sender-end of the connection. In this CNTL packet, the original KEY value and ROUTE value received from the sender-end in the FIRST packet are substituted into the CNTL packet header, both in *return* form. If the router choses to exchange ROUTE values with the sender-end, it creates a second ROUTE number, associated with the address of the destination, and includes this ROUTE value in the XROUTE field of the CNTL packet sent back to the sender node on the first network.

Once the CNTL packet arrives at the sender node, the sender adopts the router's XROUTE value, and includes the *return* form of it in subsequent packet transmissions for the given connection, in the ROUTE field. Thus, the

router may use different ROUTE values for packets traveling in different directions. Note that the routers overwrite the ROUTE field values incoming from packets generated by the other router. New ROUTE field values are generated at each hop.

Each individual route can exist for an extended period of time, (i.e., perhaps even for days inside routers.) By allowing more than one context to *share* a route, the cost of initializing and maintaining the route can be shared among contexts. Additionally, the *rate* control for the shared route can also be shared. Sharing routes allows the routers to combine redundant table entries in internal routing tables and minimize their space requirements. XTP supports route sharing, and inheritance between contexts.

In XTP, an existing route can be utilized by a newly-forming context by setting the ROUTE value in the header of the FIRST packet to the ROUTE number associated with the particular route. This number is available in the context record of any active context currently using the route.

One complication of route sharing is that the router can not detect when the route is no longer being used without being explicitly requested to *release* the route. In XTP, the special information packet type ROUTE is used by routers and nodes to tear down routes. When a node knows that it is finished using a given route, it issues a ROUTE packet to the router, which contains a RELEASE request embodied in the information segment. The router responds by issuing its own ROUTE packet acknowledging the request and releasing the route.

XTP Fragmentation Issues

XTP also supports fragmentation of data packets when necessary. The need arises when two connected networks have different *maximum transmission unit* sizes, as mentioned earlier. In this case, the routers perform the fragmentation transparently. The resulting set of smaller packets are referred to as *fragments*, although they are legitimate XTP frames themselves. Each fragment contains its own header, a portion of the original packet's data segment and its own trailer.

XTP CNTL packets are sufficiently small that they do not require fragmentation. The largest CNTL packet contains a 24 byte header, 16 byte trailer, 40 byte constant subset of the *control* segment and 16 SPAN groups containing 8 bytes each, also located in the *control* segment. The maximum number of bytes in a CNTL packet is thus 208 bytes plus the media framing.

During fragmentation, the router must refrain from exactly duplicating the original data packet's header and trailer into the smaller fragments because certain option flags are *non-replicatable*. For example, the SREQ bit in the common trailer must not be replicated — if it were, each fragment would solicit its own CNTL packet status response from the receiver, when only one was desired. Partial exceptions are the *first* fragment's header and the *last* fragment's trailer. The first fragment's header is an *exact* duplicate of the original packet's header. All other fragments contain different SEQ numbers, and perhaps other differences from the original header. The last fragment's trailer would be an exact duplicate of the original trailer *except* that the HTCHECK header-trailer checksum is calculated over a different header from the original packet's HTCHECK.

A method is under development for combining packets at a router which have identical ROUTE fields. The combined packet is referred to as a SUPER packet, and contains a special experimental header referred to as a SUPER header. The individual XTP packets can be recovered if the SUPER packet must be fragmented.

XTP Multicast Mode

XTP defines a *multicast* mode of operation where one sender can broadcast the same data stream or datagram sequence to multiple receivers simultaneously (one-to-many).

XTP's multicast mode is similar in operation to the single receiver mode in many respects. The transmitter issues a FIRST packet, and subsequent DATA packets. SREQ is used to solicit CNTL packets. *Error* control is supported using the *go-back-n* retransmission scheme; *selective retransmission* is not supported. Note that in

multicast connections the allocated buffer space in each receiver may vary in size. Essentially, data transmission proceeds at the pace of the slowest receiver.

When a multicast receiver detects out-of-sequence data, it multicasts the CNTL packet, called a *reject* packet, so that all other receivers on the connection realize that an error has occurred.

If the multicast involves a large number of receivers, the sender will be inundated with *reject* packets as all receivers clammor to announce the error. To dampen this effect, XTP requires receivers to refrain from sending the multicast *reject* packet when aware that the sender has been properly notified. The receivers monitor the network for other *reject* packets during the time the packet is being prepared and waiting for transmission. If another *reject* packet arrives, destined for the sender on the same multicast context, the receiver compares its own RSEQ value to the one contained in the newly arrived packet. RSEQ is significant because this is the next byte the multicast receivers will accept — remember, no gaps are allowed in multicast mode. If the receiver's own RSEQ number is greater than or equal to the packet's RSEQ number, the receiver refrains from sending its own *reject* packet. In this case, the rollback requested in the existing *reject* packet covers the request at the current receiver also. If, on the other hand, the receiver's own RSEQ value is smaller than the packet's, the receiver outputs its own *reject* packet. The basic idea is to guarantee reliable reception at all receivers of the data stream, without complicating the sender's task.

XTP also allows the multicast mode to operate in a less reliable "no error" mode in which receivers discard garbled packets, and inform their host of the occurrance, but no *reject* packet or retransmission scheme is used. This technique is appropriate for, say, broadcasting sensor data in a control system — the data are generated continuously, and a particular lost value is quickly replaced with a more current reading.

Prioritization Issues In XTP

XTP supports prioritization of packet processing at both the sender and receiver using *preemptive priority scheduling*. Thus, if the server is currently processing a low priority packet as a higher priority packet arrives for service, the server is *preempted* from processing the lower priority packet and begins processing the higher priority packet. Only after all higher priority packets have been completed or blocked will the server return to the low priority packet. The granularity of pre-emption (i.e., whether on a *byte*, *frame*, or *message* basis) is currently under study.

In XTP, two preemptive schedulers exist — one for incoming packets, and one for outgoing packets. For both the reader and sender prioritization schemes, XTP supports 2^{32} different priorities. Each context is associated with a particular priority level. Multiple contexts can be at the same priority level simultaneously.

For outgoing packets, the priority level is encoded into a 4 byte integer and placed into the SORT field before transmission. When the packet arrives at the remote receiver, the SORT field is examined, and the packet is enqueued according to its priority.

In XTP, the priority level is inversely proportional to the value of the integer encoding — i.e., larger SORT field values have lower priority. This scheme is static, in that the priority level remains constant as the packet travels through the network. XTP also supports a dynamic preemptive scheduling scheme based on *deadline* times and synchronized system clocks with 100 microsecond resolution. In this mode, the original SORT field value represents a future clock time (the *deadline)* whose priority is proportional to the immediacy of the deadline. As the system clock time advances towards the deadline, the packet's priority level increases. As with the static SORT mode, lower SORT values also correspond to higher priority levels, and are used to determine the queue into which the packet should be placed.

The two scheduling schemes just described are not allowed to co-exist on any given XTP network. Each network may utilize one or the other, *but not both* simultaneously. Alternatively, priority operation may be disabled altogether.

Byte Ordering

An interesting feature of the XTP packet format concerns the order in which bytes are arranged in a word for various computers. This ordering affects the sequence in which the bytes are placed onto the network. Bytes within a word can either be arranged from highest to lowest address, or from lowest to highest address. Different equipment manufacturers support different byte orderings. Since no standard exists, XTP provided a natural way to support both orderings transparently.

These two orderings are referred to as *big-endian*, and *little-endian*. In *big-endian*, the most significant byte is transmitted first. In *little-endian* the least significant byte is transmitted first. Thus *big-endian* transmits from most significant byte to least significant byte, and *little-endian* transmits vice versa.

The problem is to encode in each packet an indication of which byte ordering was used by the sender to prepare the packet, and in such a way that a receiver adhering to either byte ordering scheme can determine the correct order of the bytes. This was solved in XTP using two bit flags. The position of the two flags were chosen so that they map into each other even if the byte ordering is guessed incorrectly. The two flags are both set to the same value by the sender. These flags are called the LITTLE bits, and are found in the highest and lowest byte of the first four bytes in each packet's header segment (the *command word)*. When the LITTLE bits are set to one, the sender issued the packet using *little-endian* byte ordering. If the LITTLE bits equal zero, the packet is in *big-endian* format. If necessary, the XTP receiver process remaps each sequence of 4 bytes into the ordering preferred by its host.

More Details Available

A more complete tutorial is available from the Computer Networks Laboratory at the University of Virginia. Copies may be ordered by writing:

Professor Alfred C. Weaver
Director, Computer Networks Laboratory
Department of Computer Science
Thornton Hall
University of Virginia
Charlottesville, Virginia 22903
weaver@virginia.edu

REFERENCES

1. "XTP Protocol Definition Revision 3.4", Protocol Engines, Incorporated, 1900 State Street, Suite D, Santa Barbara, California 93101, 1989.

2. Chesson, Greg, "The Protocol Engine Project", UNIX Review, Vol. 5, No. 9, September 1987.

3. Chesson, Greg, "Protocol Engine Design", USENIX Conference Proceedings, Phoenix, Arizona, June 1987.

4. Comer, Douglas, **Internetworking with TCP/IP**, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

5. Stallings, William, **Handbook of Computer Communications Standards, Volume 1: The Open Systems Interconnection (OSI) Model and OSI-Related Standards,** Macmillan Inc., 1987.

6. Jacobsen, Van and Braden, R.T., "TCP Extensions for Long-Delay Paths", Request for Comment 1072 (RFC 1072), 1988.

7. Cheriton, David, "VMTP: Versatile Message Transaction Protocol *Protocol Specification*", Preliminary Version 0.6, Stanford University, 1988.

8. Clark, David, and Lambert, Mark, "NETBLT: A Bulk Data Transfer Protocol", Request for Comment 998 (RFC 998), 1987.

9. ISO 8348 International Organization for Standardization, **Addendum 2: Covering Network Addresses**

10. Hutchison, David, **Local Area Network Architectures**, Addison-Wesley, Wokingham, England, 1988.